REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED FINAL/01 Nov 93 TO 31 May 95

4. TITLE AND SUBTITLE ESTABLISHMENT OF A LABORATORY FOR VISUAL PROGRAMMING RESEARCH	5. FUNDING NUMBERS 4276/AS F49620-94-1-0026
6. AUTHOR(S) MASOUD T. MILANI	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SCHOOL OF COMPUTER SCIENCE FLORIDA INTERNATIONAL UNIVERSITY UNIVERSITY PARK MIAMI, FL 33199	8. PERFORMING ORGANIZATION AFOSR-TR-95 0668
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR/NM 110 DUNCAN AVE, SUTE B115 BOLLING AFB DC 20332-0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER F49620-94-1-0026
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The United States Air Force Office of Scientific Research awarded an equipment grant to Florida International University to establish a laboratory with modern workstations to facilitate research in the area of visual languages. The grant was awarded on 1 November 1993 for a period of one year.

The laboratory required a relatively large space with appropriate networking infra-structure in place. The location of the labs was secured within the University's ECS (Engineering and Computer Science) building in Summer of 1994. The purchase of required equipment were completed in Spring of 1995.

19951018 050

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR(SAME AS REPORT)

Establishment of a Laboratory for Visual Programming Research

Final Report

Funded by Air Force Office of Scientific Research

Masoud T. Milani

School of Computer Science
Florida International University
University Park
Miami, FL 33199

Tel.: (305) 348-2925
Fax: (305) 348-3549
email: milani@fiu.edu

Sponsoring Scientific Office: AFOSR/NM
110 Duncan Avenue, Suite B115
Bolling AFB DC 20332-001

Contract Number: F49620-94-1-0026DEF

1. Introduction

The United States Air Force Office of Scientific Research awarded an equipment grant to Florida International University to establish a laboratory with modern workstations to facilitate research in the area of visual languages. The grant was awarded on 1 November 1993 for a period of one year.

The laboratory required a relatively large space with appropriate networking infra-structure in place. The location of the lab was secured within the University's ECS (Engineering and Computer Science) building in Summer of 1994. The purchase of required equipment were completed in Spring of 1995.

The laboratory conducts research both at the graduate and undergraduate levels in the general area of visual languages. The main focus of our group is the application of visual techniques to Parallel Programming, Distributed Interactive 3D Applications and Virtual Reality. There are ongoing research projects that are supervised by the Principle Investigator and conducted by students. On the average, about 5 students per semester have participated. With the complete laboratory equipment in place at this time, it is expected that student participation to increase. Most of the students participate in our group for at least two semesters. The first semester is utilized to study the background literature, learn how to use various software systems and the details of existing projects. Armed with the knowledge acquired in their first semester, the students spend their second and possibly their third semester designing and implementing a software system appropriate to their level of seniority.

2. Summary of ongoing projects

This section provides a summary of various research projects that are conducted within the laboratory.

2.1 Visualization techniques for Parallel Programming

Parallel computers have traditionally provided computing power only to a limited number of (largely scientific) users. With rapid improvements in hardware technology, however, parallel computers are no longer just experimental machines in the computer science labs. The near future promises to put parallel computers on a desk top and make them available to the general public.

If the tools to assist parallel programming remain relatively primitive, such technological advances may not be fully exploited by the general computing community. But help may be available from experience with sequential programming, where visual languages have been a valuable tool for assisting software development. The value of visual programming languages and graphical interfaces has long been recognized in the area of sequential programming. We claim that bringing visual programming techniques to the domain of parallel programming will be equally valuable. Based on the idea that pictures can help both understanding and remembering, visual programming research, for examples) aims at the design and implementation of easy-to-use programming languages and tools to increase the programmer's expressive power and productivity.

Visualization research falls in two broad categories: processing visual data (images) and languages that allow programming using visual (graphical) expressions. Our research effort is concentrated on the development of visual languages.

Programming for parallel machines has been a substantial technical problem. If we wish to develop visual programming languages for parallel machines, we need a test bed for conducting experiments in language design. Such experiments might include changing the graphical representations of tasks, program data, output data, flow of control and communication, for example. If we can rapidly implement such language design, the time gap between the conception of new ideas and their validation is minimized.

To this end, our Visual Editor Generator System aims at the development of a experimental test bed for the development of visual programming languages.

To discuss the Generator system, the design language Raddle is chosen as our target parallel language. VERDI is a visual environment that allows the simulation of concurrent designs in Raddle. Our paper describes how VERDI-like interfaces for Raddle can be specified within the Generator system. The difference between VERDI and our interface is that we are using a formal framework to define the visual interface as opposed to VERDI's ad-hoc interface . The formal approach affords the language designer the possibility to modify the set of allowed editing operations and change the shapes and the placement of objects in the target visual interface at the specification level.

2.2 Distributed Interactive 3D Applications

The ability to interact with objects and applications in a three dimensional virtual environment is one of the key issues to improving human-computer interactions and hence productivity. However, the large demand on computational resources requires the components of such a system be distributed over a network of heterogeneous machine architectures.

VRAPI is a system that allows developers to build applications with a three dimensional user interface that run in a virtual environment whose components are managed by applications running on machines dispersed in a heterogenous network. Many user can simultaneously participate in this environment which allows users to interact with the various components of the environment and with each other. VRAPI is essentially a library of functions that developers use to build distributed applications with a VRUI and of a process that implements the functionality of each function in the library by accepting data connections from multiple applications and by servicing any requests in the form of messages that are generated by calls to the functions and that arrive through the connections.

This environment contains applications, each of which consisting of virtual objects whose behavior is controlled by client applications. The client applications communicate with each other and with a nexus process called the world server via the message passing mechanism. Messages are sent through reliable data connections between the client applications and the world server. Client applications can place requests with the world server to modify the properties of various objects in the virtual environment.

A key feature of the world server is to allow applications to specify that they must be notified whenever certain events occur in the virtual world. An object handled by a client application can send text descriptions of various desired constraints to the world server. This constraint is described in terms of the properties of the object placing the constraint and the properties of any other objects. The world server will check this constraint every time some change occurs in the virtual world and will send a notification message to the object that placed the constraint that the constraint has been satisfied. This constraint checking mechanism is flexible enough to allow applications to keep a consistent view of all objects in the world server database at a relatively low demand of CPU cycles and network bandwidth. (See Appendix B for details.)

3. Equipment Acquired

Quantity	Equipment	Vendor
1.	HP Deskjet 1200C Postscript Color Printer.	National Data Products
1.	4 MB simm for Deskjet 1200C	National Data Products
4.	Ink Cartridge	National Data Products
1.	P5-66 Pentium (66MHz) PC	Gateway 2000
1.	GDM-17SE1 Sony 17" Monitor	Megabyte International
2.	LO1052 Supra 28.8 External Modems	National Data Products
1.	WB-W050EX Workstation. Indigo 2 Extreme,	Silicon Graphics
	32 MB, 100/50MHz, 1.0 GB disk, 19" Monitor	Silicon Graphics
1.	W8A1-5032 Worksation, Indy, 100 MHz R4600PC 8 Bit Color, 32MB, 535 MB disk, 17" Monitor	Silicon Graphics
4.	W8B1-1G32 Workstation, Indy, 133 MHz R4600SC 8 Bit Color, 32MB, 1.0 GB disk, 17" Monitor	Silicon Graphics
1.	PC ZPD6934KS Zenith Z-Star EX DX2/50 MHz 4MB, 200MB disk, 9.5" Monochrome LCD	Zenith Data Systems
1.	AC3100 Western Digital 1 GB, Hard drive	Dataflex Corporation
2.	CE-VR Crystal Eyes Stereoscopic viewer	Stereographics Corp.
1.	Spaceware IMC; Advcd Interaction Dev. Kit; manual, driver, demos, libraries, Spaceball Model2003; Power adapter; Serial Interface Cable	Spaceball Technologies
1.	Software: SC4-Inventor-2.0 Iris Open Inventor	Silicon Graphics
1.	Software: SC4-Perf-1.2 Iris Performer Real-Time Graphics Dev. Environment	Silicon Graphics
1.	Software: SC4-DMDEV-2.0 Digital Media Dev. Software Libraries	Silicon Graphics

4. Student Participation

As of Summer of 1995, the following students have been members of our research group and have utilized the laboratory equipment to carry out their work.

- Carry Bakker
- Carlos Flores
- Louis Florit
- Amyn Gilani
- Andre Henry
- Virgiliue Mocanu
- Alfredo Rojas
- Johan Sosa
- Cory Tsang
- Kendrick Vargas
- Holger Wiechert
- Christopher Wong

Appendix A.

A Test Bed for Experimenting with Visualization of Parallel Programs

F. Arefi, M. Evangelist and M. Milani
School of Computer Science
Florida International University
University Park
Miami, FL 33199
arefi@fiu.edu

Abstract

Because of the lack of the software tools to assist concurrent programming, programming for parallel computers has been a significant technical problem for diverse range of users. We are concentrating on techniques that allow computing and non-computing experts to define what they need and then automatically generate the specified visual language. Consequently, our visual language research aims at the development of a test bed for conducting experiments in language design and speed up the implementation process for tools to assist parallel computing.

1 Introduction

Parallel computers have traditionally provided computing power only to a limited number of (largely scientific) users. With rapid improvements in hardware technology, however,

> parallel computers are no longer just experimental machines in the computer science labs. The near future promises to put parallel computers on a desk top and make them available to the general public [6].

If the tools to assist parallel programming remain relatively primitive, such technological advances may not be fully exploited by the general computing community. But help may be available from experience with sequential programming, where visual languages have been a valuable tool for assisting software development. The value of visual programming languages and graphical interfaces has long been recognized in the area of sequential programming. We claim that bringing visual programming techniques to the domain of parallel programming will be equally valuable [7, 3]. Based on the idea that pictures can help both understanding and remembering, visual programming research (see [4, 5], for examples) aims at the design and implementation of easy-to-use programming languages and tools to increase the programmer's expressive power and productivity.

Visualization research falls in two broad categories: processing visual data (images) and languages that allow programming using visual (graphical) expressions. Our research effort is concentrated on the development of visual languages.

Programming for parallel machines has been a substantial technical problem. If we wish to develop visual programming languages for parallel machines, we need a test bed for conducting experiments in language design. Such experiments might include changing the graphical representations of tasks, program data, output data, flow of control and communication, for example. If we can rapidly implement such language design, the time gap between the conception of new ideas and their validation is minimized.

To this end, our Visual Editor Generator System aims at the development of a experimental test bed for the development of visual programming languages.

To discuss the Generator system, the design language Raddle [9] is chosen as our target parallel language. VERDI [12] is a visual environment that allows the simulation of concurrent designs in Raddle. Our paper describes how VERDI-like interfaces for Raddle can be specified within the Generator system. The difference between VERDI and our interface is that we are using a formal framework to define the visual interface as opposed to VERDI's ad-hoc interface . The formal approach affords the language designer the possibility to modify the set of allowed editing operations and change the shapes and the placement of objects in the target visual interface at the specification level.

2 The Visual Editor Generator

The Visual Editor Generator System [1, 2] provides a formal framework to visually specify syntax-directed diagram editors and automatically generate them from specifications. The family of diagram editors that may be specified and generated within the editor generator system are essentially customized graph editors that share common structures and functionalities. A graph editor might be customized in two primary ways. Customizing the *physical appearance* of the graphs being manipulated involves declaring different node shapes and colors, edge shapes and colors and the geometric relationships among them. The set of possible *graph manipulations* (editing operations) are customized by providing a collection of graph transformation rules that allow or disallow editing operations in certain contexts.

The editor generator system allows generating editors that are customizable in both the physical appearance of graphs and the editing operations that are allowed. Customizability of the physical appearance is addressed by providing general SmallTalk classes for graph nodes and edges that might be specialized (inherited from) to capture the application specific behavior of various node and edge types. Within the framework of such customizations one would specialize the general node and edge classes to specific subclasses that contain information regarding the desired shape, color, and so on. This is similar to Edge [11] which facilitates graph editor customization by providing general C++ node and edge classes. The graph editing operations are customized by describing the allowable context-dependent graph editing operations using a starting graph and a collection of specialized graph transformation rules [1].

A customized editor designer develops a specialized graph editor by providing the generator system the specification of the needed editor. This specification contains the description of both the desired *physical appearance* and *graph manipulations.* The generator supports the development of both parts of the specification by providing appropriate tools. The Generator system provides the editor designer with four editors. The *Set Editor* allows the declaration of the labels of nodes and edges of the underlying graph transformation system. The *Graph Editor* provides facilities to develop and specify the target editor's underlying starting graph and transformation rules. The *Image Editor* allows the development of icons associated with various node, edge and rule types. The *Relation Editor* is used to specify the geometric relationships among node types. Once the specification of the target editor is complete, the generator system automatically produces the target editor. Figure 1 depicts an overall view of the generator system.

The visual language specification is a form of graph grammar that is customized for the specification of syntax-directed visual editors. Graph grammars are two dimensional in nature, are inherently concurrent, and allow for the dynamic creation and deletion of nodes and arcs, and, therefore, are suitable for the specification of visual parallel programming languages [8, 10].

3 Formal Specification of Visual Languages

In [1] we introduced a unified framework to specify visual languages and their permitted manipulations. This framework allows languages be defined as one initial object and a collection of editing operations. This way, any object that can be obtained by applying a sequence of allowed editing operations to the initial object is defined to be a program in the language being specified. That is, languages are defined as dynamic objects that, as the result of applying editing operations, change from one form to another with each form being a member of the language.

3.1 Graph Transformation System

The Visual Editor Generator's underlying specification framework is a graph transformation system[1]. A Graph Transformation System is a 5-tuple

$$T = < L_V, L_E, S, EO, M >$$

where

- L_V and L_E are sets of symbols called the node label set and the edge label set, respectively,
- S, the starting graph, is a labeled directed graph over $< L_V, L_E >$,
- EO is a collection of editing operation labels and
- M is a collection of graph transformation rules.

A graph transformation system defines a graphical language and its syntax directed editor. It has an

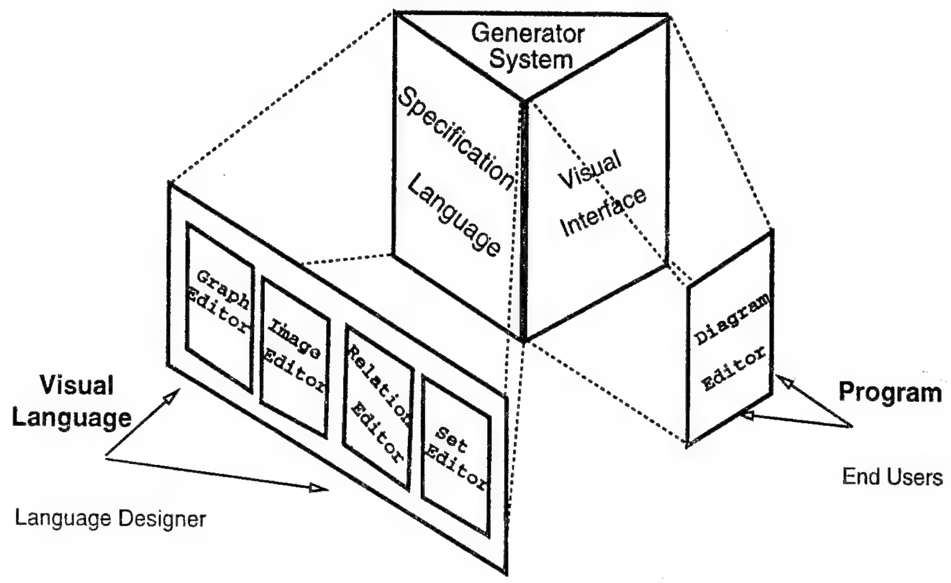


Figure 1: The Generator System

internal memory that is organized as a graph which initially contains the starting graph. Once a graph is in the internal memory, the graph transformation system repeatedly applies its transformation rules until no rule can be applied.

To discuss the specification language as well as the Generator system, the design language Raddle [9] is chosen as our target parallel language.

4 Raddle a Design Language

Raddle is a robust linguistic tool suitable for designing large, concurrent software systems. Raddle permits the designer to develop a large software system at a high level of abstraction by partitioning the system into "teams" of asynchronous processes that coordinate and communicate with each other. The designer can then decompose the interactions into equivalent and smaller interactions. (See [9] for more details.)

4.1 Concepts and Abstraction

VERDI [12] is a visual environment that allows the simulation of concurrent designs in Raddle. This section provides an overview of the underlying concepts for VERDI-like interface.

4.1.1 The Primitive Actions

A VERDI interface is designed from five primitive actions, Figure 2 depicts the icons.

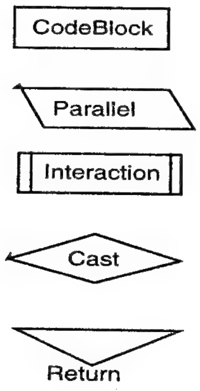


Figure 2: **The Primitives**

- Code-block: is a set of assignment statements.
- Parallel: consists of a set of statements that are executed in parallel.
- Interaction: is the synchronization and communication mechanism.
- Cast: transfers control to procedures.
- Return: passes control to the point that the procedure was invoked.

4.1.2 Control Flow

There are three control constructs:

- Choice: selection of one of the several actions.
- Sequence: execution of actions sequentially left to right.
- Iterator: iteration of several actions.

4.1.3 Parallel Activities

A Team consists of set of communicating Roles, which may be either Processes or Procedures. Synchronization and communication among the roles in a team is accomplished by n-parity interaction. (See [9] and [12] for complete and detailed information.)

5 The Specification of VERDI-like language

The specification of VERDI-like interface consists of description of both the *visual appearance* and *graph manipulations*. The generator system supports the development of both parts of the specification by providing appropriate tools.

- Step1: defining the nodes and edge labels.
  - Node labels: Verdi, Team, Role, Proc(for process), Prod(for procedure) Action, End, Body(for the body of a proc or prod), Selbody(for the expanding to more choices), Codeblock, Parallel, Interaction, Cast, Return and Lambda (for defining constraints).
  - Editing operation labels: Mycursor (for cursor position), Start, Addteam, Addteamafter, Addrole, Addroleafter, Addprocess, Addprocedure, delete and etc.
  - Edge labels: Apply, Nteam (link between teams), Bteam (for going back to previous team, we use this kind of edge for geometric attribute evaluation), Nrole, Brole, Ctf (control flow), Bctf (backward control flow), Itr (iteration), Cff (control flow false), Bcff and etc.

  The language designer uses the *Set* editor, *Image* editor and the *Relation* editors to specify the labels of the nodes and edges, to develop the shapes associated with each node and edge, and also to specify the geometric relationship among nodes respectively.

- Step2: defining the start graph.
  Start graph is the smallest meaningful program in the language. Figure 3 depicts the start graph for VERDI-like interface.

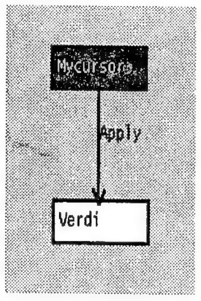


Figure 3: **The start graph**

- Step3: defining the transformation rules for the language.
  The transformation rules are functions that can be applied to graph and modify it accordingly. Figures 4 and 5 depicts the left hand side and a right hand side of a graph transformation rule, respectively.

The language designer uses the generator's *Graph* editor to specify the *start graph* and the *transformation rules*.

5.1 An Example of Applying a Transformation Rule to a Graph

Figures 4 and 5 depict a graph transformation rule for expanding a node named *Body* to *iteration construct*. Figure 4 is the left side of the rule and it models a graph that this rule can be applied to. The condition here is that the *cursor* should highlight a node named *Body* and the *Body* should have an incoming edge labeled *Ctf* from a node (it can be any node) and it should have an outgoing edge labeled *Bctf* to the same node. Also *Body* should have an outgoing edge labeled *Ctf* to a node and should have an incoming edge labeled *Bctf* from the same node. If there is a match between the left rule and the graph then the transformation rule is applicable to the graph. The following steps are taken for applying a transformation rule to a graph.

- Delete all the left hand side's pattern nodes (rectangular shaped nodes) from the graph. In our

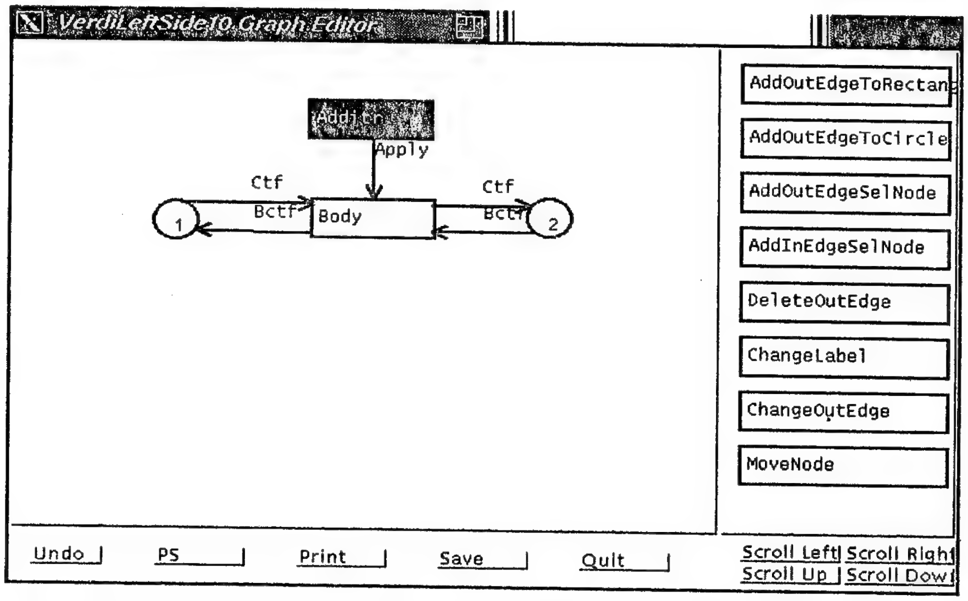


Figure 4: **The left side of a transformation rule for add iteration**

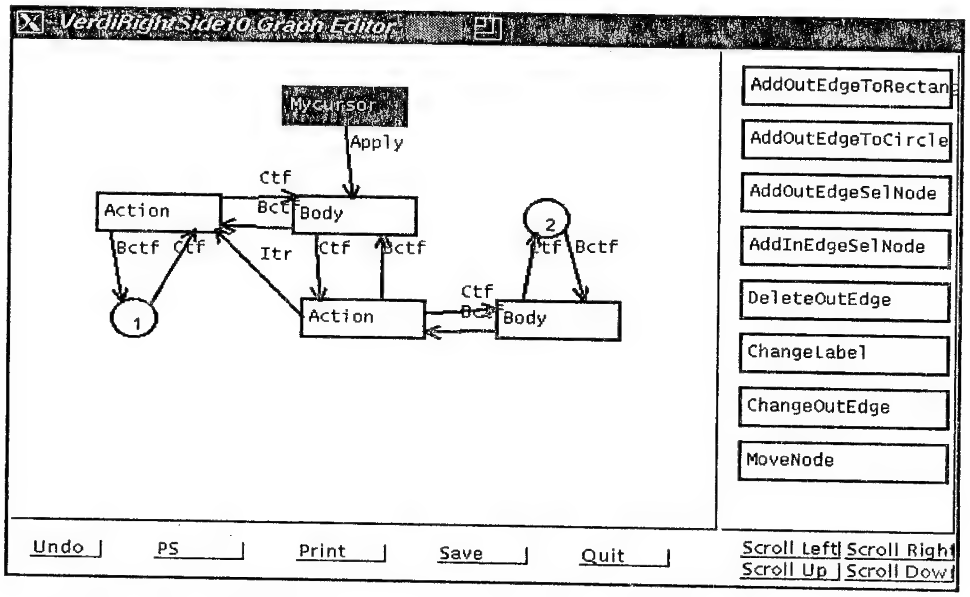


Figure 5: **The right side of a transformation rule for add iteration**

example the nodes *Additr* and *Body* get deleted from the graph.

- Add all the right hand side's pattern nodes (rectangular shaped nodes) to the graph and then add the new edges. In our example *Action, Body, Action, Body* and corresponding edges get added to the graph.
- Highlight the node being pointed at by *Mycursor*.
- Find all the applicable rules for the highlighted node and display those in the menu area.

Figure 6 shows the VERDI-like language before applying the Additr command. Figure 7 depicts the the language after application of the command.

6 The Generation of VERDI-like Interface (language)

Appendix A depicts some of the rules for VERDI-like language. After the specification of all the nodes and edges and transformation rules, the Generator Accepts all the information and generates the Visual Syntax-directed editor for the target language. Figure 8 shows the visual syntax-directed editor interface for VERDI-like language. Figures 8 and 9 depict the high level design of an electronic funds transfer system. Because of the limitation of the screen space the last team of eft-System has been shown in Figure 9. (For more detail [12]).

7 Conclusion

In this paper we explained the specification and the generation of the VERDI-like language and its interface. The difference between VERDI and our interface is that we are using a formal framework to define the visual interface as opposed to VERDI's ad-hoc interface. The formal approach affords the language designer the possibility to modify the set of allowed editing operations and to change the shapes and the placement of objects in the target visual interface at the specification level. The Generator system provides user friendly tools that could be used during and after the generation of any new visual language. These tools will help the language designer to develop their new ideas through see and feel process. They can easily change the functionality and physical appearance of their language and get the immediate feedback by using one of the generator's tools and consequently the implementation effort for parallel computation tools is reduced. Using our tool, we have been able to generate the VERDI-like interface in a few week.

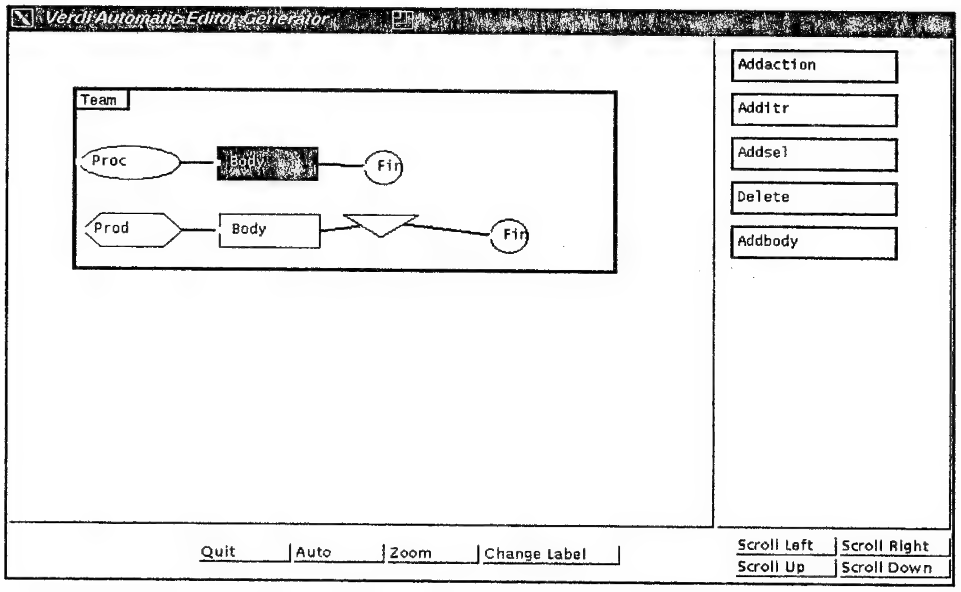


Figure 6: **The screen before applying Additr**

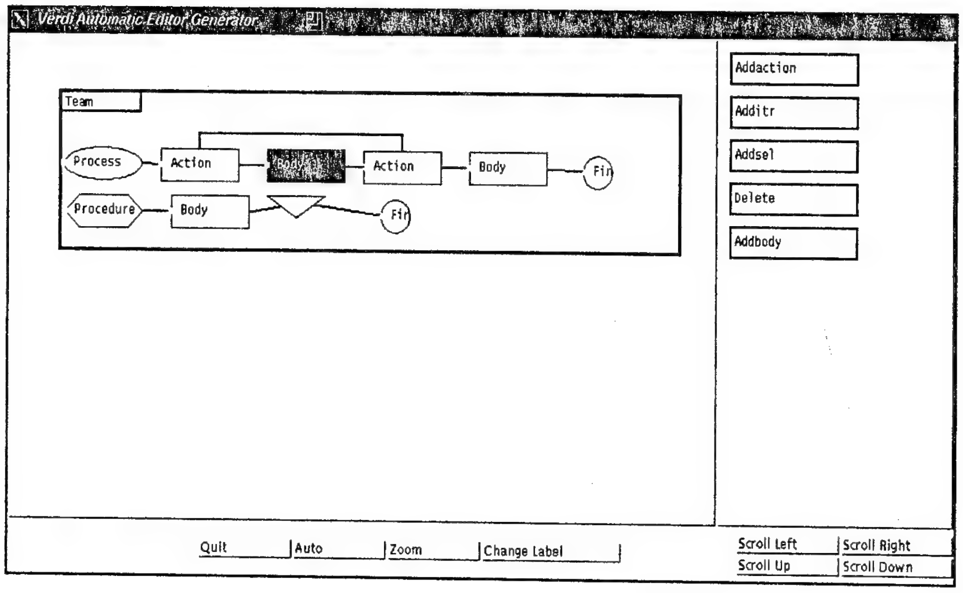


Figure 7: **The screen after applying Additr**

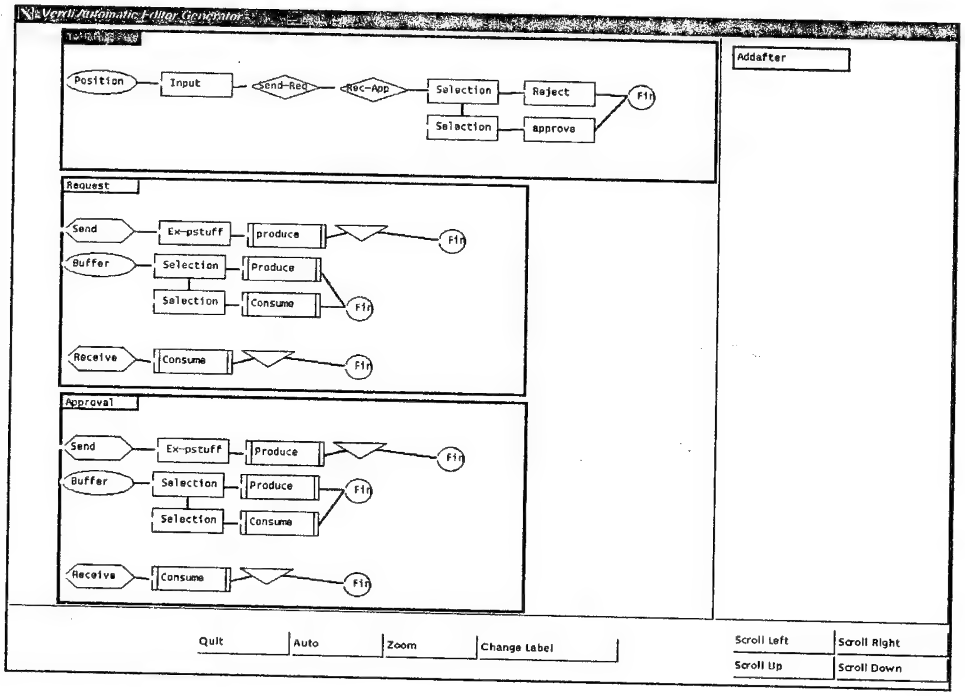


Figure 8: **High Level Design for Electronic Funds Transfer System**

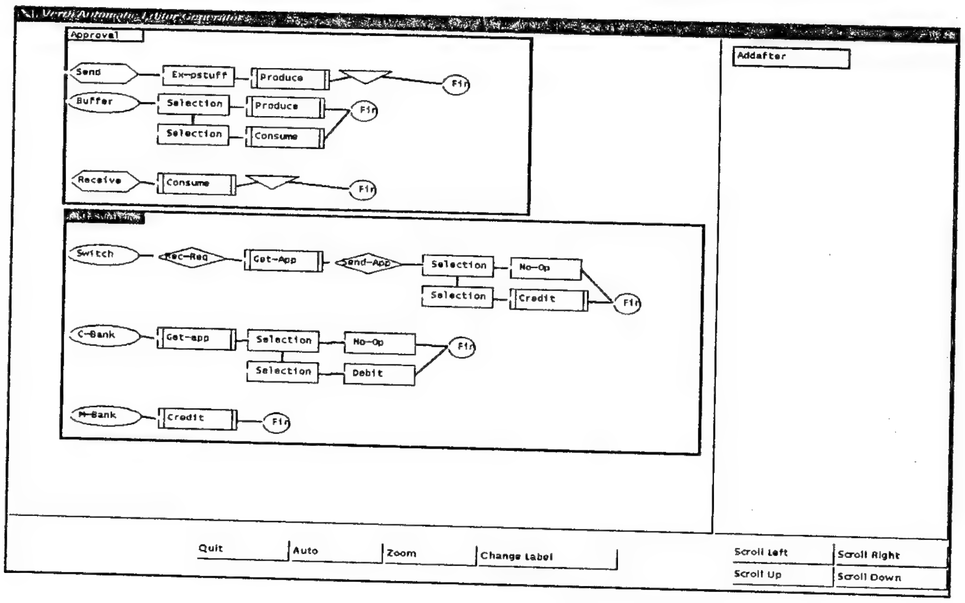
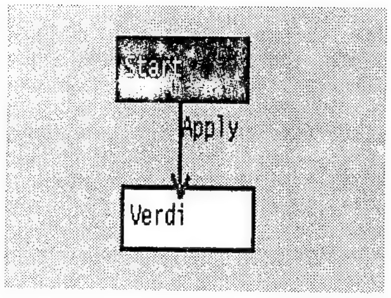
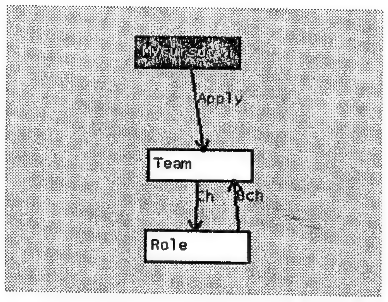
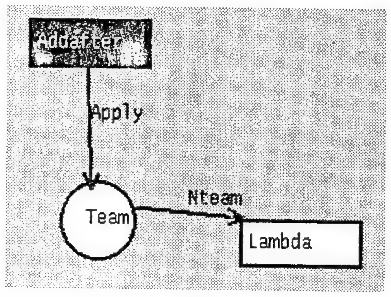


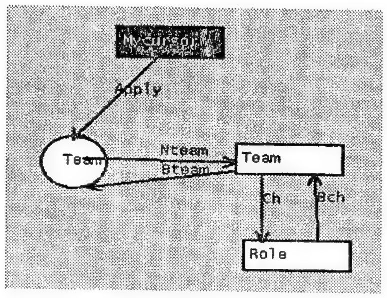
Figure 9: **eft-system continue**

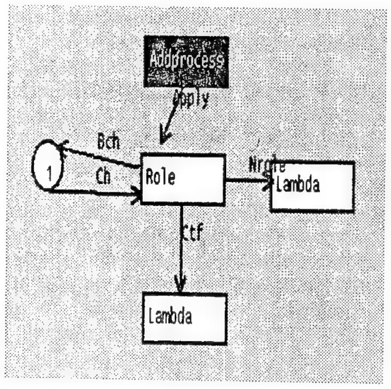
8 Appendix A: Graph Transformation Rules for VERDI-like Language

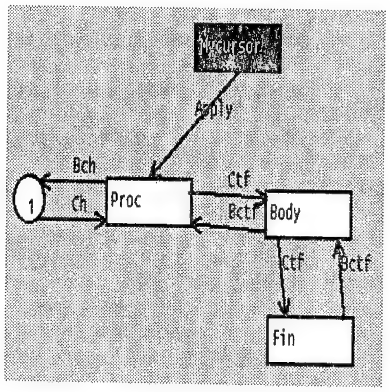


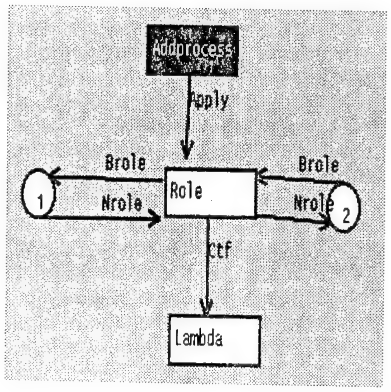


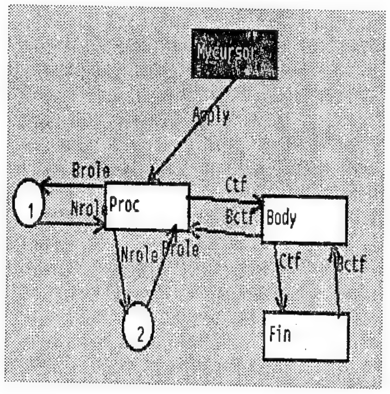


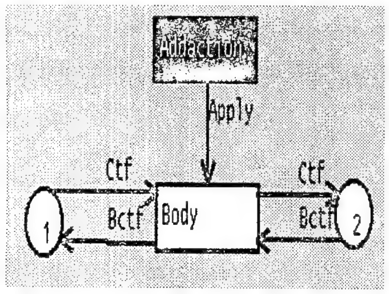


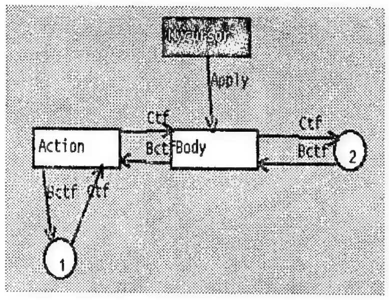


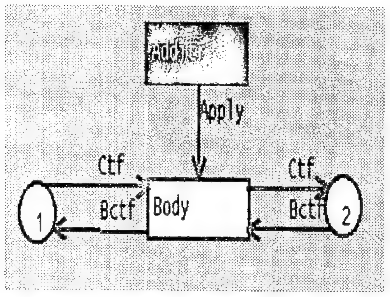


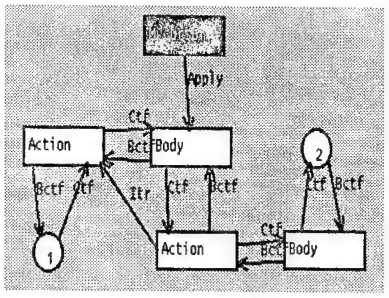


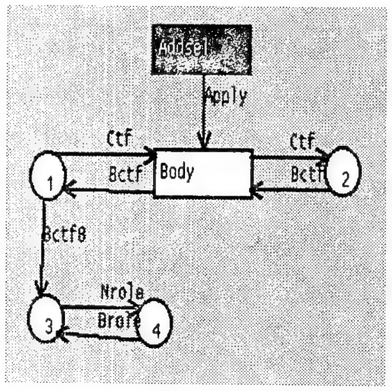


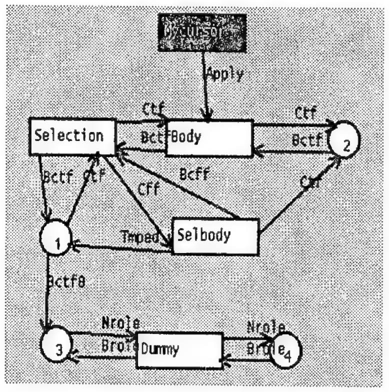

Addaction
Apply
Ctf
Bctf
Body
Ctf
Bctf
1
2

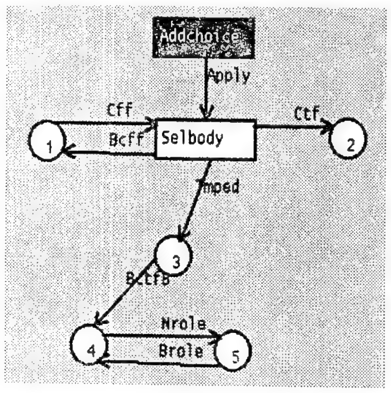

Nycursor
Apply
Action
Ctf
Bctf
Body
Ctf
Bctf
2
Bctf
Ctf
1

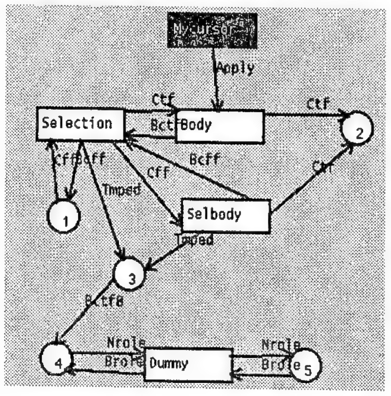

Additer
Apply
Ctf
Bctf
Body
Ctf
Bctf
1
2


Apply
Ctf
Action
Bctf
Body
2
Bctf
Ctf
Itr
Ctf
Bctf
Ctf
Bctf
1
Ctf
Action
Bctf
Body


Addsel
Apply
Ctf
Bctf
Body
Ctf
Bctf
1
2
Bctf8
Nrole
Brole
3
4


Nycursor
Apply
Ctf
Ctf
Selection
Bctf
Body
Bctf
2
Bctf
Ctf
Bcff
Cff
Ctf
Tmped
Selbody
1
Bctf8
Nrole
Nrole
3
Brole
Dummy
Brole
4


Addchoice
Apply
Cff
Bcff
Selbody
Ctf
1
2
Tmped
3
Bctf8
Nrole
Brole
4
5


Nycursor
Apply
Ctf
Ctf
Selection
Bctf
Body
2
Cff
Bcff
Bcff
Cff
Tmped
Ctf
1
Selbody
Tmped
3
Bctf8
Nrole
Nrole
4
Brole
Dummy
Brole
5

References

[1] F. Arefi, C. Hughes, and Workman D. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3), 1990.

[2] F. Arefi and M. Milani. Generating diagram-editors from formal specifications. *Journal of Information and Software Technology*, 34(3):139–146, 1992.

[3] E. Glinert, M. Blattner, and C. Frerking. Visual tools and languages: Directions for the 90's. In *Proceedings of 1991 Workshop on Visual Languages (VL'91)*, pages 89–95, October 1991.

[4] E.P. Glinert, editor. *Tutorial–Visual Programming Environments; Volume 1 : Paradigms and Systems*, Washington, D.C., 1990. IEEE Computer Society Press.

[5] E.P. Glinert, editor. *Tutorial–Visual Programming Environments; Volume 2 : Applications and Issues*, Washington, D.C., 1990. IEEE Computer Society Press.

[6] J. Hartman and D. Sanders. Teaching a course in parallel processing with limited resources. *SIGCSE Bulletin*, 23(1):97–101, March 1991.

[7] E. Levin. Grand challenges to computational science. *CACM*, 32(12):1456–1457, December 1989.

[8] J.P. Loyall. Specification of concurrent programs using graph grammars. In *PhD thesis*. University of Illinois, 1991.

[9] Evangelist M., Shen V.Y., Forman I.R., and Graf M. Using raddle to design distributed systems. In *Proceedings of the 10th International Conference on Software Engineering*, April 1988.

[10] M. Nagl and A. Schurr. A specification environment for graph grammars. In H. Ehrig, H. J. Kreowski, and G. Rozenberg, editors, *Proceedings of 4th International Workshop on Graph Grammars and Their Application to Computer Science, LNCS 532*, pages 599–609. Springer-Verlag, 1991.

[11] F. N. Paulisch and W. F. Tichy. Edge: An extendaible graph editor. *Software Practice & Experience*, 20(S1):63–88, June 1990.

[12] v. y. Shen, C. Richter, M. Graf, and J. Brumfield. Verdi: A visual environment for designing distributed systems. *Journal of Parallel and Distributed Computing*, 9:128–137, 1990.

Appendix B.

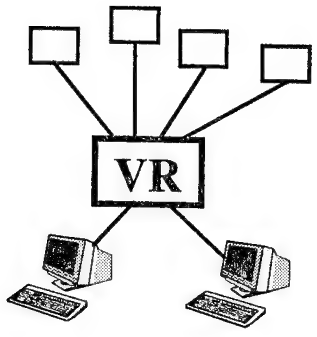
Programmer's Manual

VRAPI: Virtual Reality Application Programming Interface for a Network Distributed Sharable Virtual Reality Environment

Version 1.0

by

Virgiliu Mocanu
mocanuv@solix.fiu.edu



Florida International University
School of Computer Science

System Overview

VRAPI is a programming library used for developing applications that run on different machines on a network and that have a Virtual Reality user interface. A set of applications running together constitute a virtual environment that can be shared by more than one user at the same time. Through this environment, users can interract with each other through the use of applications developed with VRAPI such as a virtual telephone or virtual bodies, etc.

VRAPI is a set of functions written in C. The Reference Manual describes the functionality and syntax of these function calls. The Reference Manual also tells the developer if the functions are blocking or non-blocking since the applications written with these function calls are supposed to function in a distribuited shared environment and preferably have real-time response.

The system consists of three general groups of components. The first group is the set of clients or applications. Examples of these applications are a virtual terminal for running a shell, an virtual airplane for flight simulation, a CAD application for engineers, etc. These clients are separate processes running on arbitrary machines on a network. The clients manage virtual objects. A set of virtual objects managed by one or more clients work together to form an application. For example, a virtual airplane application might consist of the wings, fuselage, tail, landing ghear, control pannel, etc. All these objects can be controlled by one single client or if the developer chooses to really modularize things, he may want to program one client that implements the behavior of the objects that make up the body of the airplane (wings, tail, fuselage) and a different client that implements the behavior of the control pannel and instruments of the airplane. These two cseparate clients may communicate with each other via message passing by using the ICCM (Inter Client Communication Mechanism) feature of VRAPI.

It is important to understand the distinction between the client and the object. A client is a **process** that runs on some machine on the network. It contains **code** that causes some virtual object in the VR world to change its properties such as position, rotation, shape, etc. A client is like a pupeteer controlling the virtual objects. A client may manage more than one object so the client may contain code that implements the behavior of more than one object or even more than one kind (class) of object. For example, an application simulating the furniture inside a virtual house may manage a cocktail table, a couch, and two chairs. These 4 objects are examples of 3 different classes of objects: 1 of class table, 1 of class couch, and 2 of class chair.

The next component in the system is the world server. This is a single process that stores information about all objects in the virtual world. This is basically a database to which clients connect and of which they make requests to modify the objects in the world. For example clients may connect to the world server, may request the creation of some buildings, a floor, some trees, etc. Other clients may request the creation of some cars and later may periodically request the movement of the cars.

The last general group of components that make up the system are the display clients. These are processes that also connect to the world server and are updated by the world server with the location, shape, etc, of every object in the world. The display clients then render a scene of these objects as seen from some point of view (usually the user's location). Obviously, every user must run at least one display client in order to get feedback from the virtual world. The display clients or some other processes in the same group as the display clients read input from some input device such as a mouse or Spaceball and change the user's location and orientation in the virtual world.

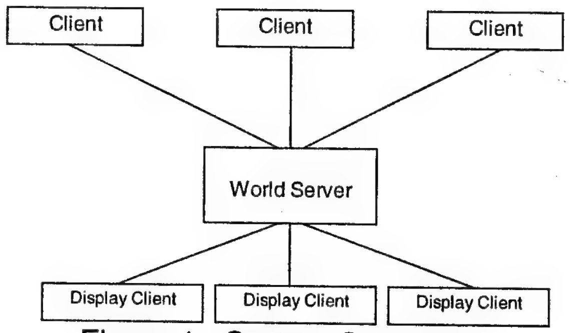


Figure 1 - System Schematic

Figure 1 shows how the three types of components are linked to each other and how they communicate. Basically, all communication is done with or through the world server.

Architecture of a Client

The structure of a client is somewhat different than the structure of a regular program that is running independent of any other program (i.e. a sort routine that takes data, processes it, and outputs it, and finally exits). A client written with VRAPI is an application that responds to events and messages comming from other applications or from the world server. Thus clients are passive programs that are dormant when no messages are present and only process whenever something "interesting" happens and the world server tells them about it.

As we mentioned in the previous section, the client contains code that handles the behavior of an object, multiple, objects, and multiple classes of objects. This code resides inside the body of different functions.

Figure 2 shows a graphical representation of a client's control flow. VRAPI contains functions for the first two steps in this diagram. Naturally, the user must provide code for the functions in steps 3, 4, and 5. These functions implement the behavior of different classes of objects. They determine what kind of message they are handling (usually with a "switch" statement) and then send messages back to the world server requesting that the objects behave in a certan way such as move, rotate, scale, change shape, etc.

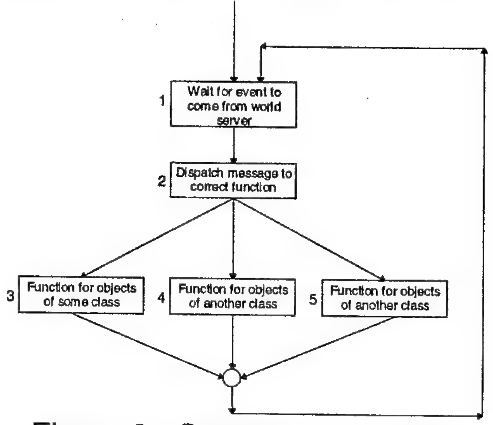

1
Wait for event to come from world server
2
Dispatch message to correct function
3
Function for objects of some class
4
Function for objects of another class
5
Function for objects of another class

Figure 2 - Structure of a client

So far we have shown how clients receive messages, dispatch them to the right functions, and process them inside the functions. We now need to know how the whole structure of the client looks and we need a more detailed description of each step.

In step 0, the client initializes any global variables and creates the objects by calling the "CreateObject" function provided by VRAPI. The function "CreateObject" asks for the function address that implements the behavior of the object (supplied by you) as well as the object class number returned by "RegisterClass." The function "CreateObject" returns an object number that is unique in the virtual world. "CreateObject" stores this number in a database and associates it with the function address.

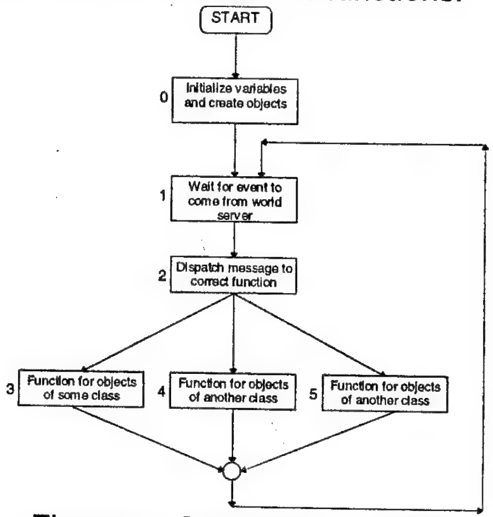

START
0
Initialize variables and create objects
1
Wait for event to come from world server
2
Dispatch message to correct function
3
Function for objects of some class
4
Function for objects of another class
5
Function for objects of another class

Figure 3 - Structure of a client

Step 1 is a call to "ReadMessage" provided by VRAPI which blocks until a message is available from the world server and which returns a pointer to a message structure. This message structure contains all the necessary information for the processing of the message. This information includes the object number that should receive the message.

Step 2 is a call to “DispatchMessage” which takes as input the pointer to the message structure and calls the function supplied by you which implements the behavior of the object. This is done by looking up the object number (present in the message structure) in the database and retreiving the function number (which was stored when you called “CreateObject”).

Steps 3 to 5 are functions supplied by the developer. These functions are celled with several parameters. One of them is the pointer to the message structure returned by “ReadMessage.” The other parameters are redundant information already present in the message structure.

The body of a function that implements the behavior of an object usually consists of a “switch” statement that looks at the message structure and determines what kind of message was received by an object. You can then determine which object received the message by also looking inside the structure. This way different objects (of the same class) can behave differently.

Object Behavior Function

The object behavior function defines the behavior of a certain class of objects. It is called by VRAPI whenever a client receives a message for the object associated with that object behavior function. The association is made when calling "CreateObject." The object behavior object is called from within the "DispatchMessage" function.

Strictly speaking, the object behavior function handles messages before they are sent to the server. This means that whenever an object is sent a message such as the MOVE_OBJECT message which specifies the new location of an object in 3D, the client responsible for that object receives the message, then dispatches it to the object behavior function. The object behavior function then decides what to do with this message. It can either call the function "DefaultServerAction" which basically sends the message back to the world server in which case the world server implements the moving of the object by changing that object's (x, y, z) coordinates in its world database. Alternatively, the object behavior function can choose to ignore the MOVE_OBJECT message or perhaps take a different action altogether by calling "RotateObject" or such.

Figure 4 shows sample code for an object behavior function. This function is called from within "DispatchMessage" with 3 parameters. The **first** is a pointer to a data structure containing information about the current connection between the client and the world server. The **second** is the object number that the message is for. The **third** is a pointer to the message structure (which redundantly contains the object number as one of its fields).

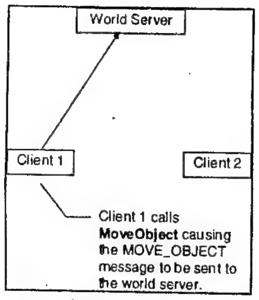
```
void ObjBehaviorFn(PCONNECTION pConn,
                   HOBJECT hObj,
                   PMESSAGE pMsg) {

  switch(pMsg->iMsgType) {
  case INITIALIZE_OBJECT:
    /* This is the first message sent to this
       function after the object is created in
       the world server
    */
    break;

  case MOVE_OBJECT:
    break;

  default:
    printf("Don't know how to handle this\n");
    break;
  }
  DefaultServerAction(pConn, pMsg);
}
```

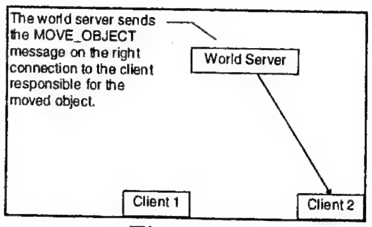
Figure 4 - Sample object behavior function

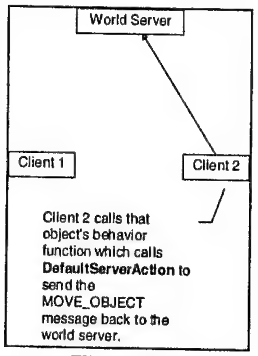
The body of the function is simple: **First** figure out which message is being handled and take appropriate action. **Next** call "DefaultServerAction" to send the message back to the world server and let it take the appropriate action. In this example, the function "DefaultServerAction" is called in all situations. You may want to call "DefaultServerAction" only for some messages while for others the message should just be "swallowed" by your function.

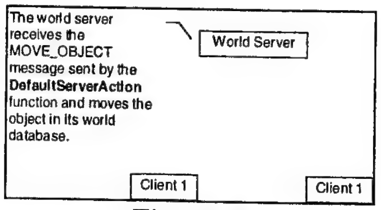
To sumarize the message passing scheme, Figures 5-8 show the steps that are taken behind the scenes between the time that a client calls the function "MoveObject" and the time that the object is actually moved.

A small axception occurs when a client calls "MoveObject" to move its own object. In this situation, the client automatically sends the message dirrectly to itself instead of going through the world server.


World Server
Client 1
Client 2
Client 1 calls **MoveObject** causing the MOVE_OBJECT message to be sent to the world server.
Figure 5


The world server sends the MOVE_OBJECT message on the right connection to the client responsible for the moved object.
World Server
Client 1
Client 2
Figure 6


World Server
Client 1
Client 2
Client 2 calls that object's behavior function which calls **DefaultServerAction** to send the MOVE_OBJECT message back to the world server.
Figure 7


The world server receives the MOVE_OBJECT message sent by the **DefaultServerAction** function and moves the object in its world database.
World Server
Client 1
Client 1
Figure 8

Object Shape Formats

Virtual objects have various characteristics. An object has fundamental characteristics such as position, orientation, scale, etc. Similarly, an object has visual characteristics such as shape, color, texture, transparency, luminosity, etc. These characteristics are stored in the server by the clients that create the objects. Later, when the objects need to be displayed by some user's viewer application, the object shape is downloaded from the server by calling "GetShape" and is rendered by the viewer.

The shape of an object can be specified in many ways. Among the most commonly used formats for specifying 3D object shapes are DXF, 3DS, etc. VRAPI does not place any constraint on the object shape but leaves this decision up to the implementor of the viewer application. VRAPI allows this flexibility by permiting the application to register the shape description for an object in more than one format. For example, an application creating a cube can call "DefineShape" or "DefineShapeFromFile" with different format names and different format bodies as their parameters.

```
DefineShapeFromFile(pConnection, hCube, "VOF", "cube.vof");
```

and

```
DefineShapeFromFile(pConnection, hCube, "DXF", "cube.dxf");
```

This way two distinct descriptions for the same object are stored in the world server. One description is in the VOF format while the other is in the DXF format. Later a viewer application can call "GetShapeFormat" to enumerate all available formats and once it finds a format that it can interpret, it can call "GetShape" to download the description in that particular format.

This mechanism allows clients to specify the shape of an object in various degrees of detail complexity. The viewer application can then download only the format that it can actually render in real time.

A Simple Application

In this section we will see a sample application that loads an object from a file and creates it in the world. The following listing shows the code:

cube.c

```
/*
    cube.c  -   an application that connects to the server
                loads the object in file 'cube.vof' and does
                nothing else
*/

#include <stdio.h>
#include <vrlib.h>

HOBJECT hObj1, hLast;

void RegisterShape(PCONNECTION, HOBJECT, char *, char *);
void fnObjFunction1(PCONNECTION, HOBJECT, PMESSAGE);

/************************************************************/
void usage_error() {
  fprintf(stderr,
          "Usage: cube machine\n");
  exit(1);
}
/************************************************************/
main(int argc, char **argv) {
  PCONNECTION pConnection;
  char server_hostname[100];
  PMESSAGE pMsg;
  HCLASS hClass1;

  if(argc < 2) {
    usage_error();
  }
  strcpy(server_hostname, argv[1]);

  pConnection = ConnectToServer(server_hostname);
  if(pConnection == (PCONNECTION) -1) {
    fprintf(stderr,
          "Could not connect to server on machine %s\n",
          server_hostname);
    exit(1);
  }
  else {
    printf("Connection made OK\n");
  }

  hClass1 = RegisterClass(pConnection, "TestClass");

  hObj1 = CreateObject(pConnection, hClass1, NULL,
                       fnObjFunction1,
                       0, 100, 0, 0, 0, 0);
  printf("Sending object shape...\n");
  DefineShapeFromFile(pConnection, hObj1, "VOF", "cube.vof");
  printf("Done sending object shape...\n");

  while(pMsg = ReadMessage(pConnection)) {
    DispatchMessage(pConnection, pMsg);
  }

  DisconnectFromServer(pConnection);
}
/************************************************************/
void fnObjFunction1(PCONNECTION pConn, HOBJECT hObj,
                    PMESSAGE pMsg) {
  /*
     This function is just a pass-through function that
     defaults every message back to the world server.
  */
  DefaultServerAction(pConn, pMsg);
}
/************************************************************/
```

cube.vof

```
OBJECT
  TRIANGLE <0.000000 -0.000000 0.000000> <10.000000 -0.000001
0.000000> <10.000000 -0.000001 10.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <0.000000 -0.000000 0.000000> <10.000000 -0.000001
10.000000> <0.000000 -0.000001 10.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <0.000000 -0.000000 0.000000> <0.000000 -0.000001
10.000000> <0.000000 10.784311 10.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <0.000000 -0.000000 0.000000> <0.000000 10.784311
10.000000> <0.000000 10.784313 -0.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <0.000000 -0.000001 10.000000> <10.000000 -0.000001
10.000000> <10.000000 10.784311 10.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <0.000000 -0.000001 10.000000> <10.000000 10.784311
10.000000> <0.000000 10.784311 10.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <10.000000 -0.000001 10.000000> <10.000000 -0.000001
0.000000> <10.000000 10.784312 -0.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <10.000000 -0.000001 10.000000> <10.000000 10.784312
-0.000000> <10.000000 10.784311 10.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <10.000000 -0.000001 0.000000> <0.000000 -0.000000
0.000000> <0.000000 10.784313 -0.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <10.000000 -0.000001 0.000000> <0.000000 10.784313 -
0.000000> <10.000000 10.784312 -0.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <10.000000 10.784311 10.000000> <10.000000 10.784312
-0.000000> <0.000000 10.784313 -0.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
OBJECT
  TRIANGLE <0.000000 10.784311 10.000000> <10.000000 10.784311
10.000000> <0.000000 10.784313 -0.000000>
  COLOR <1.000000 1.000000 1.000000>
ENDOBJECT
```

The main program of **client.c** is just an initialization section followed by the infinite loop that reads messages from the queue and dispatches them to the object behavior function. The object behavior function does not do anything special. It does not handle any messages but simply passes them on to the world server to be handled in a default manner.

The listing of **cube.vof** shows the definition of the shape of the object being displayed. The cube is made up of 12 triangles (two per face). The color of every triangle is white (RGB of 1, 1, 1). The object shape definition is given to the server with the function "DefineShapeFromFile" which simply reads the file and sends the format to the world server. This way, any display client can download the shape in order to render an image of the object.

Because the object behavior function does not do anything special, the object can be moved, rotated, etc. by any other application. For example, the users of the VR environment may be running clients that allow them to grab objects and move them around. When the object is sent a MOVE_OBJECT or ROTATE_OBJECT message by these clients, it simply passes them on to the server which then moves or rotates the object.

The client is run in the following way: Assume you have a world server running on some machine called **frontier** and a display client which is connected to the world server on **frontier**. In order to run the application, compile it

```
% cc -o cube cube.c -lvr -Llibrary_path -Iinclude_path
```

and run it

```
% cube frontier
```

Automatic Constraint Checking Mechanism

The VRAPI world server is equiped with an automatic constraint checking mechanism that checks if certain constraints registered by clients have been met every time something in the system changes and if the constraints are met, then the client that registered the constraints is notified with a message.

Constraint checking by the world server simplies the design of the clients and reduces their computational load. For example, when implementing a push button for some VRUI (Virtual Reality User Interface), the button object "wants to know" when some user pointer is touching it so that it might perform some action like change position and notify the application to which it belongs.

In order to do something like this, the client managing the button would have to send to the world server the constraint specifications. When the user pointer moves close to the button, the constraint would be satisfied which would cause the world server to send a message to the object behavior function of the button object.

In order to register the constraint, the client would call "DefineConstarint" with the handle of the button, the value of the event that the world server will send to the object when the constraint is satisfied, and the ASCII text specifying the constraint. This specification would look something like this:

```
1  distance = ((other_absolute_x - current_absolute_x)^2 +
2               (other_absolute_y - current_absolute_y)^2 +
3               (other_absolute_z - current_absolute_z)^2)^(0.5)
4  @ (distance < 10) && (other_class == 2)
```

The first 3 lines calculate the distance between the button object (current object) and any other generic object (other object). Line 4 begins with the character '@' which indicates a constraint. This line checks if the generic object is within 10 units of the button object and if the generic object is of class 'user pointer' which is predefined as value 2.

Constraints can be formulated in terms of any object characteristic. Each object characteristic has a value which can be accessed by the constraint definition through variable names. A complete list of all the variable names can be found in Appendix A.

When the world server checks a particular constraint and determines that the constraint has been satisfied, the world server sends a message to the object behavior function of the object that placed the constraint. The message value is the one specified when the constraint was defined and the message body contains details about the other (generic) object that caused the constraint to be satisfied. This information includes things such as the other object's class, handle, position, orientation, scaling, etc. Appendix A contains a full description of the information contained in the message body.

A Client With Constraints

In this example we will see how a client can use the automatic constraint checking mechanism to change position. We will develop a client that creates an object which starts to spin on its Y axis whenever a user pointer moves within its proximity. The client simply places a constraint with the world server to notify the object whenever another object of class "user pointer" is within the proximity of the object.

When this happens, the world server will send the notification message to the object behavior function of our client. The client will respond by calling "RotateObject" and increasing the object's rotation around the Y axis.

proxim.c

```
/*
    proxim.c - an application that connects to the server
               loads the object from file 'cube.vof' and causes
               the object that it creates to spin if a user
               pointer moves within its proximity
*/

#include <stdio.h>
#include <vrlib.h>
#include <math.h>
#include "proxim.h"

HOBJECT hObj1, hLast;
FLOAT4  rotation_y = 0.0;

void RegisterShape(PCONNECTION, HOBJECT, char *, char *);
void fnObjFunction1(PCONNECTION, HOBJECT, PMESSAGE);

/*************************************************************/
void usage_error() {
  fprintf(stderr,
          "Usage: cube machine\n");
  exit(1);
}
/*************************************************************/
main(int argc, char **argv) {
  PCONNECTION pConnection;
  char server_hostname[100];
  PMESSAGE pMsg;
  HCLASS hClass1;

  if(argc < 2) {
    usage_error();
  }
  strcpy(server_hostname, argv[1]);

  pConnection = ConnectToServer(server_hostname);
  if(pConnection == (PCONNECTION) -1) {
    fprintf(stderr,
            "Could not connect to server on machine %s\n",
            server_hostname);
    exit(1);
  }
  else {
    printf("Connection made OK\n");
  }

  hClass1 = RegisterClass(pConnection, "TestClass");

  hObj1 = CreateObject(pConnection, hClass1, NULL,
                       fnObjFunction1,
                       0, 100, 0, 0, 0, 0);
  printf("Sending object shape...\n");
  DefineShapeFromFile(pConnection, hObj1, "VOF", "cube.vof");
  printf("Done sending object shape...\n");

  if(!SetupMyConstraints(pConn, hObj)) {
     fprintf(stderr, "Could not register constraint\n");
    exit(1);
  }

  while(pMsg = ReadMessage(pConnection)) {
    DispatchMessage(pConnection, pMsg);
  }

  DisconnectFromServer(pConnection);
}
/*************************************************************/
```

```
/*************************************************************/
void fnObjFunction1(PCONNECTION pConn, HOBJECT hObj,
                    PMESSAGE pMsg) {

  switch(pMsg->iMsgType) {
  case PROXIM_NOTIFY:
    RotateObject(pConn, hObj, 0, rotation_y, 0);
    rotation_y += 5;
    return;
  }
  DefaultServerAction(pConn, pMsg);
}/*************************************************************/
```

proxim.h

```
#define PROXIM_NOTIFY (USER_MESSAGES + 0)

HCONSTRAINT SetupMyConstraints(PCONNECTION, HOBJECT);
```

constraints.c

```
HCONSTRAINT SetupMyConstraints(PCONNECTION pConn,
                               HOBJECT hObj) {
  char b[1000];
  b[0] = 0;

  sprintf(&b[strlen(b)],
          "d = sqr((current_absolute_x-other_absolute_x)^2 +");
  sprintf(&b[strlen(b)],
          "        (current_absolute_y-other_absolute_y)^2 +");
  sprintf(&b[strlen(b)],
          "        (current_absolute_z-other_absolute_z)^2)");
  sprintf(&b[strlen(b)],
          "@ (d < 100) && (other_class == %d)", USER_POINTER);
  return(DefineConstraint(pConn, hObj, PROXIM_NOTIFY, b));
}
```

The program consists of two C files and a header file. You must compile and link the program

```
% cc -o proxim proxim.c constraints.c -lvr -Llibrary_path -Iinclude_path
```

and run it

```
% proxim serverhostname
```

You must also run a viewer and an application that lets the user interract with the VR world through a pointer object of class USER_POINTER. You must then move this pointer in the proximity of the cube to see the cube spin on its axis.

Appendix A: Automatic Constraint Checking Mechanism

The automatic constraint mechanism is a feature of the world server that allows clients to register constraints which are checked by the server every time a change occurs in the world. When a constraint is satisfied, the object associated with that constraint is sent the notification message whose value is associated with the constraint.

In order to register the constraint, a client must call the function "DefineConstraint." The three important arguments passed to the function are

1. **Handle of object that places the constraint and that should be notified when the constraint is satisfied.**
2. **Value of the notification message that will be sent to the object's behavior function. This value must be greater than or equal to the value of USER_MESSAGES.**
3. **A pointer to a string containing the constraint specifications as ASCII text.**

The body of the constraint specification consists of ASCII text. The grammar of the language is shown here:

```
language             →    statament_list condition_list
                        | condition_list
statament_list       →    statament_list statament
                        | statament
statament            →    VARIABLE = expression
expression           →    function ( expression )
                        | function2 ( )
                        | expression + expression
                        | expression - expression
                        | expression * expression
                        | expression / expression
                        | expression ^ expression
                        | - expression
                        | ( expression )
                        | VARIABLE
                        | NUMBER
                        | expression || expression
                        | expression && expression
                        | expression == expression
                        | expression != expression
                        | expression >= expression
                        | expression > expression
                        | expression <= expression
                        | expression < expression
function             →    cos
                        | sin
                        | acos
                        | asin
                        | exp
                        | log
                        | abs
                        | sqr
condition_list       →    condition_list condition
                        | condition
condition            →    @ expression
function2            →    shape_change_in_other
                        | created_other
                        | destroyed_other
VARIABLE             →    any legal C variable name
NUMBER               →    any integer or floating point number
```

Bold letters indicate actual characters while normal letters indicate expansions.

The constraint specifications are interms of two objects: The first object is the one that is registering the constraint and is called the “current object” while the second is any generic object and is called the “other object.” The constraint specifications can be constructed out of predefined variables which are assigned numeric values corresponding with the characteristics of the objects being checked. Table A1 lists all of these predefined variables.

Generic Object	Current Object	Interpretation of Field's Value
other_object	current_object	handle number
other_class	current_class	class number
other_relative_x	current_relative_x	coordinates ralative to parrent before object is transformed to world coordinate system
other_relative_y	current_relative_y	
other_relative_z	current_relative_z	
other_old_relative_x	current_old_relative_x	old relative coordinates before last change
other_old_relative_y	current_old_relative_y	
other_old_relative_z	current_old_relative_z	
other_absolute_x	current_absolute_x	coordinates in world coordinates after object is transformed to world coordinate system
other_absolute_y	current_absolute_y	
other_absolute_z	current_absolute_z	
other_old_absolute_x	current_old_absolute_x	absolute coordinates before last change (may be used to calculate velocity)
other_old_absolute_y	current_old_absolute_y	
other_old_absolute_z	current_old_absolute_z	
other_rotation_x	current_rotation_x	rotation around the axes in parent's coordinate system
other_rotation_y	current_rotation_y	
other_rotation_z	current_rotation_z	
other_old_rotation_x	current_old_rotation_x	rotation around axes before last change (may be used to calculate angular velocity)
other_old_rotation_y	current_old_rotation_y	
other_old_rotation_z	current_old_rotation_z	
other_scaling_x	current_scaling_x	scaling
other_scaling_y	current_scaling_y	
other_scaling_z	current_scaling_z	
other_old_scaling_x	current_old_scaling_x	old scaling (may be used to calculate scaling velocity)
other_old_scaling_y	current_old_scaling_y	
other_old_scaling_z	current_old_scaling_z	
other_front_x	current_front_x	unit vector that points along the positive z axis in parent coordinate system before rotation may point in a different location after rotation
other_front_y	current_front_y	
other_front_z	current_front_z	
other_old_front_x	current_old_front_x	front vector before last change
other_old_front_y	current_old_front_y	
other_old_front_z	current_old_front_z	
other_up_x	current_up_x	unit vector that points along the positive y axis in parent coordinate system before rotation may point in a different location after rotation
other_up_y	current_up_y	
other_up_z	current_up_z	
other_old_up_x	current_old_up_x	up vector before last change
other_old_up_y	current_old_up_y	
other_old_up_z	current_old_up_z	
other_right_x	current_right_x	unit vector that points along the positive x axis in parent coordinate system before rotation may point in a different location after rotation
other_right_y	current_right_y	
other_right_z	current_right_z	
other_old_right_x	current_old_right_x	right vector before last change
other_old_right_y	current_old_right_y	
other_old_right_z	current_old_right_z	

Table A1

The constraint specification language contains pre-defined functions. Table A2 lists all these functions.

Function	Parapeter	Description
cos	radians angle	returns the cosine of the given angle
sin	radians angle	returns the sine of the given angle
acos	ratio	calculates the inverse cosine function of the given number and returns the angle in radians
asin	ratio	calculates the inverse sine function of the given number and returns the angle in radians
exp	float	raises the natural constant e to the given number
log	float	calculate sthe natural logarithm of the given number
abs	float	returns the absolute value of the given number
sqr	float	returns the square root of the given number
shape_change_in_other()	none	returns 1 if the other (generic object has changed shape), else returns 0
created_other()	none	returns 1 if the other object was just created, else returns 0
destroyed_other()	none	returns 1 if the other object was just destroyed, else returns 0

Table A2

When the world server checks a constraint and determines that the constraint placed by an object has been met by some other generic object, the first object is sent a notification message with the message value equal to that specified when the constraint was placed and the message mody containing various information about the other (generic) object that satisfied the constraint. This information is shown in Table A3.

Byte Position	Format (Big Endian)	Description
0	4 byte integer	handle of other object
4	4 byte integer	handle of other object's parent (NULL if no parent)
8	4 byte integer	class of other object
12	4 byte floating point number	absolute x of other object
16	4 byte floating point number	absolute y of other object
20	4 byte floating point number	absolute z of other object
24	4 byte floating point number	x component of front pointing vector in other object
28	4 byte floating point number	y component of front pointing vector in other object
32	4 byte floating point number	z component of front pointing vector in other object
36	4 byte floating point number	x component of up pointing vector in other object
40	4 byte floating point number	y component of up pointing vector in other object
44	4 byte floating point number	z component of up pointing vector in other object
48	4 byte floating point number	x component of right pointing vector in other object
52	4 byte floating point number	y component of right pointing vector in other object
56	4 byte floating point number	z component of right pointing vector in other object
60	4 byte floating point number	relative x coordinate of other object
64	4 byte floating point number	relative y coordinate of other object
68	4 byte floating point number	relative z coordinate of other object
72	4 byte floating point number	rotation around the x axis of other object
76	4 byte floating point number	rotation around the y axis of other object
80	4 byte floating point number	rotation around the z axis of other object
84	4 byte floating point number	scaling in the x direction of other object
88	4 byte floating point number	scaling in the y direction of other object
92	4 byte floating point number	scaling in the z direction of other object

Table A3

This information can be extracted from the message by calling "ExtractNetworkData" whose functionality is similar to that of "sscanf." By using this function, the data can be converted to the format of the machine on which the client is running (from Big Endian to Little Endian perhaps).

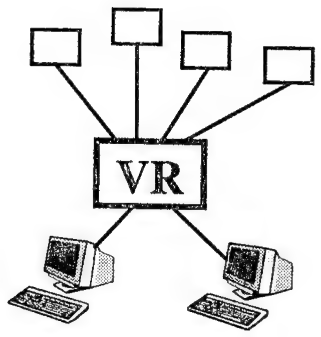
Reference Manual

VRAPI: Virtual Reality Application Programming Interface for a Network Distributed Sharable Virtual Reality Environment

Version 1.0

by

Virgiliu Mocanu
mocanuv@solix.fiu.edu



Florida International University
School of Computer Science

ReadMessage

Summary

Blocking

```
PMESSAGE ReadMessage(pConnection);
PCONNECTION     pConnection;
```

Description

ReadMessage fetches the first message in the message queue and returns a pointer to the **MESSAGE** structure. If no messages are present in the message queue associated with this connection, **ReadMessage** will block until a complete message arrives on the connection.

ReadMessage decodes all the fields except the data field in the **MESSAGE** structure from network format to machine format. The information in the data field of the **MESSAGE** structure can be extracted by calling the **ExtractNetworkData** function.

ReadMessage is normally used in conjunction with **DispatchMessage**.

Return Value

ReadMessage returns a pointer to a **MESSAGE** structure.

See Also

DispatchMessage, CreateObject

DispatchMessage

Summary

Blocking

void DispatchMessage(pConnection, pMessage);
PCONNECTION pConnection;
PMESSAGE pMessage;

Description

DispatchMessage takes a pointer to a **MESSAGE** structure (usually returned by **ReadMessage**) and dispatches it to the function that implements the behavior of an object.
DispatchMessage accomplishes this as follows:

> **DispatchMessage** first looks at the destination object handle inside the **MESSAGE** structure and then calls the appropriate function that was associated with that object when the **CreateObject** function was called.

Return Value

None.

See Also

ReadMessage, CreateObject

DefaultServerAction

Summary

Non-Blocking

```
void DefaultServerAction(pConnection, pMessage);
PCONNECTION     pConnection;
PMESSAGE        pMessage;
```

Description

DefaultServerAction sends the message pointed to by **pMessage** to the World Server on the connection pointed to by **pConnection**.

DefaultServerAction is usually called from within a function that implements the behavior of an object. Normally, this function is first called by **DispatchMessage**. The function then looks at the message and takes appropriate action(s). The function then calls **DefaultServerAction** right before exiting.

When the World Server receives the message, the World Server takes an appropriate default action in response to the message. For example, if the message DESTROY_OBJECT is received by the object function, the function can first take appropriate action and then call **DefaultServerAction** which sends the message to the World Server and takes the default action of removing this object from its database.

Return Value

None.

See Also

CreateObject, DispatchMessage, ReadMessage

SendMessage

Summary

Non-Blocking

```
void SendMessage(pConnection, iMsgType, iNetTrips, hDestObject, sData, iSize);
PCONNECTION     pConnection;
INT4            iMsgType;
INT4            iNetTrips;
HOBJECT         hDestObject;
void            *sData;
INT4            iSize;
```

Description

SendMessage creates a **MESSAGE** structure and sends it out to the network. **SendMessage** encodes for network transmission all the fields in the **MESSAGE** structure except for the message data. The message information must be encoded into a buffer by calling **FormatNetworkData** function, before calling **SendMessage**.

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** data structure returned by **ConnectToServer**.
iMsgType	**INT4**	This is a number identifying the message to be sent. i.e. DESTROY_OBJECT.
iNetTrips	**INT4**	This is the number of times a message has been passed between a Client and the World Server. When calling **SendMessage**, this number should normally be 1.
hDestObject	**HOBJECT**	This is the handle of the object to which the message should be sent.
sData	**void ***	This is a pointer to a buffer containing the data of the message in network format. The data can be encoded from machine format to network format by calling the function **FormatNetworkData**.
iSize	**INT4**	This is the size of the data in the buffer.

Return Value

None.

See Also

ConnectToServer, ExtractNetworkData, FormatNetworkData

FormatNetworkData

Summary

Non-Blocking

```
int FormatNetworkData(sFormat, sBuffer, param1, param2, param3, ..., paramN);
char          *sFormat;
void          *sBuffer;
void          *param1;
void          *param2;
void          *param3;
...
void          *paramN;
```

Description

FormatNetworkData encodes values from machine format to network (machine independent) format which may be sent to the World Server by calling the **SendMessage** function.

The parameters of this function are:

sFormat	**char ***	This is a pointer a character string that specifies what data must be converted to network format. The string contains a series of tokens separated by spaces with the following meaning: i1 data of size 1 such as **INT1** or **char** i2 data of size 2 such as **INT2** i4 data of size 4 such as **INT4** or **HOBJECT** f4 floating point value of size 4 such as **FLOAT4** f8 floating point value of size 8 such as **FLOAT8** s*N* string of characters of size *N* Each of these tokens corresponds to one of the values pointed to by **param1** through **paramN** to be converted from machine format to network format.
sBuffer	**void ***	This is the are of memory where the formatted data should be stored. This argument is later passed to **SendMessage**.
param1	**void ***	Pointer to first parameter to be converted.
param2	**void ***	Pointer second parameter to be converted.
param3	**void ***	Pointer to third parameter to be converted.
paramN	**void ***	Pointer to Nth parameter to be converted.

Return Value

Number of parameters successfully converted.

See Also

ExtractNetworkData, SendMessage

ExtractNetworkData

Summary

Non-Blocking

```
int ExtractNetworkData(sFormat, sBuffer, param1, param2, param3, ..., paramN);
char                *sFormat;
void                *sBuffer;
void                *param1;
void                *param2;
void                *param3;
...
void                *paramN;
```

Description

ExtractNetworkData decodes values from network (machine independent) format to machine format which may be used by the application. This function is normally called when handling a message that is received by the function implementing the behavior of an object.

The parameters of this function are:

sFormat	**char ***	This is a pointer to a character string that specifies what data must be converted to machine format. The string contains a series of tokens separated by spaces with the following meaning: i1 data of size 1 such as **INT1** or **char** i2 data of size 2 such as **INT2** i4 data of size 4 such as **INT4** or **HOBJECT** f4 floating point value of size 4 such as **FLOAT4** f8 floating point value of size 8 such as **FLOAT8** s*N* string of characters of size *N* Each of these tokens corresponds to one of the memory locations pointed to by **param1** through **paramN** into which network formatted data pointed to by **sBuffer** is to be converted to machine format.
sBuffer	**void ***	This is the are of memory where the formatted data should be stored. This argument is later passed to **SendMessage**.
param1	**void ***	Pointer to memory location into which first data item is to be placed after being converted from network format to machine format.
param2	**void ***	Pointer to memory location into which second data item is to be placed after being converted from network format to machine format.
param3	**void ***	Pointer to memory location into which third data item is to be placed after being converted from network format to machine format.
paramN	**void ***	Pointer to memory location into which Nth data item is to be placed after being converted from network format to machine format.

Return Value

Number of parameters successfully converted.

See Also

FormatNetworkData, SendMessage

CreateObject

Summary

Blocking

```
HOBJECT CreateObject(pConnection, hClass, hParentObj, pFunction);
PCONNECTION     pConnection;
HCLASS          hClass;
HOBJECT         hParentObj;
PFN             pFunction;
```

Description

CreateObject sends a message to the World Server requesting that a new object be created. This function blocks and waits for an acknowledgment from the World Server. The acknowledgment contains the handle of the new object which is returned to the caller if the function if successful. If the function fails, **CreateObject** returns NULL.

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** structure returned by **ConnectToServer**.
hClass	**HCLASS**	This is the handle to a class returned by **RegisterClass**, **GetClass**, or **GetObjectClass**.
hParentObj	**HOBJECT**	This is the handle of the object of whom the current object wants to be a child. If the current object needs to be a top level object, this parameter is NULL.
pFunction	**PFN**	Pointer to a function implementing the behavior of this object. **CreateObject** associates this function with this object handle so that **DispatchMessage** calls this function if the destination object handle in a message is equal to the object handle of this object.

Return Value

Handle of new object is success. NULL if failure.

See Also

ConnectToServer, GetClass, GetObjectClass, DispatchMessage, RegisterClass

DestroyObject

Summary

Non-Blocking

HOBJECT DestroyObject(pConnection, hObject);
PCONNECTION pConnection;
HOBJECT hObject;

Description

DestroyObject sends a DESTROY_OBJECT message to the object specified by **hObject** through the World Server.

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** structure returned by **ConnectToServer**.
hObject	**HOBJECT**	This is the handle of the object that will receive the DESTROY_OBJECT message. That object will only be destroyed if it calls **DefaultServerAction** with the DESTROY_OBJECT message.

Return Value

None.

See Also

ConnectToServer, DefaultServerAction, SendMessage

RegisterClass

Summary

Blocking

HCLASS RegisterClass(pConnection, sClassName);
PCONNECTION pConnection;
char *sClassName;

Description

RegisterClass requests that a new class with the name specified in **sClassName** be created in the server. If a class by that name already exists, the server sends back the handle of the class. If the class does not exist, then a new class is created with the name specified in **sClassName** and a new handle is assigned to it which is returned to the caller.

The maximum number of classes that a client can register is MAX_CLASSES_PER_CLIENT.

Two classes are predefined in the server. They are:

"**ObservationPointClass**"	with a handle value of 1
"**UserPointerClass**"	with a handle value of 2

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** structure returned by **ConnectToServer**.
sClassName	**char ***	This is a pointer to a string of characters containing the name of the class to register with the server.

Return Value

Handle of class. NULL if failure.

See Also

ConnectToServer, GetClass, GetObjectClass

GetClass

Summary

Blocking

```
HCLASS GetClass(pConnection, sClassName);
PCONNECTION     pConnection;
char            *sClassName;
```

Description

GetClass requests the handle of the class with the name specified in **sClassName.**

Two classes are predefined in the server. They are:

"ObservationPointClass"	with a handle value of 1
"UserPointerClass"	with a handle value of 2

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** structure returned by **ConnectToServer**.
sClassName	**char ***	This is a pointer to a string of characters containing the name of the class to get handle for.

Return Value

Handle of class. NULL if failure.

See Also

ConnectToServer, GetObjectClass, RegisterClass

GetClass

Summary

Blocking

```
HCLASS GetClass(pConnection, sClassName);
PCONNECTION     pConnection;
char            *sClassName;
```

Description

GetClass requests the handle of the class with the name specified in **sClassName**.

Two classes are predefined in the server. They are:

"**ObservationPointClass**"	with a handle value of 1
"**UserPointerClass**"	with a handle value of 2

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** structure returned by **ConnectToServer**.
sClassName	**char ***	This is a pointer to a string of characters containing the name of the class to get handle for.

Return Value

Handle of class. NULL if failure.

See Also

ConnectToServer, GetObjectClass, RegisterClass

GetObjectClass

Summary

Blocking

```
HCLASS GetObjectClass(pConnection, hObject);
PCONNECTION    pConnection;
HOBJECT        hObject;
```

Description

GetObjectClass requests the class handle of an object specified in **hObject**.

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** structure returned by **ConnectToServer**.
hObject	**HOBJECT**	This is the object handle for whom the class handle is requested.

Return Value

Class handle of the object specified in hObject. NULL if failure.

See Also

ConnectToServer, GetClass, RegisterClass

DefineConstraint

Summary

Blocking

HCONSTRAINT DefineConstraint(pConnection, hObject, iMsgType, sText);
PCONNECTION pConnection;
HOBJECT hObject;
INT4 iMsgType;
char *sText;

Description

DefineConstraint sends to the World Server some text specifying the circumstances under which an object should be notified by the World Server. The notification is in the form of a message of type **iMsgType** that is sent from the World Server to the object whose handle is given in **hObject**.

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** structure returned by **ConnectToServer**.
hObject	**HOBJECT**	Handle of the object that should be notified if the constraint is met.
iMsgType	**INT4**	The value of the message that will be sent to the object if the constraint is met.
sText	**char ***	This points to a buffer containing the text that defines the constraint. For the exact syntax see the **User's Guide**.

Return Value

Handle of the newly created constraint in the server. NULL if failure.

See Also

ConnectToServer, CreateObject, DestroyConstraint

DestroyConstraint

Summary

Blocking

```
void DestroyConstraint(pConnection, hObject, hConstraint);
PCONNECTION     pConnection;
HOBJECT         hObject;
HCONSTRAINT     hConstraint;
```

Description

DestroyConstraint sends a DESTROY_CONSTRAINT message to the object whose handle is in **hObject**. Upon receipt, the object may call **DefaultServerAction** with the DESTROY_CONSTRAINT message which will then cause the World Server to remove the constraint from the object's list of constraints that define when that object should be sent a notification message.

The object whose handle is specified in **hObject** must have first registered a constraint with the server by calling **DefineConstraint**.

The parameters of this function are:

pConnection	**PCONNECTION**	This is a pointer to a **CONNECTION** structure returned by **ConnectToServer**.
hObject	**HOBJECT**	Handle of the object that owns the constraint.
hConstraint	**HCONSTRAINT**	The handle of the constraint registered by the object in **hObject** with **DefineConstraint**.

Return Value

Handle of the newly created constraint in the server. NULL if failure.

See Also

ConnectToServer, CreateObject, DestroyConstraint

MoveObject

Summary

Non-Blocking

void MoveObject(pConnection, hObject, relative_x, relative_y, relative_z);
PCONNECTION pConnection;
HOBJECT hObject;
FLOAT4 relative_x, relative_y, relative_z;

Description

MoveObject sends a MOVE_OBJECT message to the object whose handle is in **hObject**. Upon receipt, the object may call **DefaultServerAction** which will then cause the World Server to change the object's (x, y, z) relative coordinates in the database.

The (x, y, z) coordinates are relative to the origin of the parent object. If the object is a top level object then the (x, y, z) coordinates represent relative coordinates with respect to the world's origin.

The absolute coordinates of the object are calculated as a function of this object's (x, y, z) coordinates, (x, y, z) rotation, as well as the parent's rotation. The World Server keeps a hierarchical structure of objects. In order to calculate the absolute coordinates of an object, the world server starts by translating and rotating first the leaves of the object tree and moves up the tree to the root constantly applying the translate/rotate process. (This is not the actual algorithm however).

A single object is first translated to its relative (x, y, z) coordinates which the function **MoveObject** modifies and then the object is rotated around the Z axis, followed by a rotation around the X axis, followed by a rotation around the Y axis by the (x, y, z) rotation values which the function **RotateObject** modifies. The absolute coordinates of all the children of this object are subsequently modified.

Return Value

None.

See Also

RotateObject, MoveRotateObject

RotateObject

Summary

Non-Blocking

```
void RotateObject(pConnection, hObject, rotation_x, rotation_y, rotation_z);
PCONNECTION     pConnection;
HOBJECT         hObject;
FLOAT4          rotation_x, rotation_y, rotation_z;
```

Description

RotateObject sends a ROTATE_OBJECT message to the object whose handle is in **hObject**. Upon receipt, the object may call **DefaultServerAction** with the message which will then cause the World Server to change the object's (x, y, z) rotations in the database.

The object's absolute coordinates are calculated as a function of the object's (x, y, z) rotation which this function modifies and (x, y, z) coordinates.

The absolute coordinates of the object are calculated as a function of this object's (x, y, z) coordinates, (x, y, z) rotation, as well as the parent's rotation. The World Server keeps a hierarchical structure of objects. In order to calculate the absolute coordinates of an object, the world server starts by translating and rotating first the leaves of the object tree and moves up the tree to the root constantly applying the translate/rotate process. (This is not the actual algorithm however).

A single object is first translated to its relative (x, y, z) coordinates which the function **MoveObject** modifies and then the object is rotated around the Z axis, followed by a rotation around the X axis, followed by a rotation around the Y axis by the (x, y, z) rotation values which the function **RotateObject** modifies. The absolute coordinates of all the children of this object are subsequently modified.

Return Value

None.

See Also

MoveObject, MoveRotateObject

MoveRotateObject

Summary

Non-Blocking

```
void MoveRotateObject(pConnection, hObject, relative_x, relative_y, relative_z, rotation_x, rotation_y,
                      rotation_z);
PCONNECTION    pConnection;
HOBJECT        hObject;
FLOAT4         relative_x, relative_y, relative_z;
FLOAT4         rotation_x, rotation_y, rotation_z;
```

Description

MoveRotateObject sends a MOVE_ROTATE_OBJECT message to the object whose handle is in **hObject**. Upon receipt, the object may call **DefaultServerAction** with the message which will then cause the World Server to change the object's (x, y, z) relative coordinates and (x, y, z) rotations in the database.

The object's absolute coordinates are calculated as a function of the object's (x, y, z) rotation which this function modifies and (x, y, z) coordinates.

The absolute coordinates of the object are calculated as a function of this object's (x, y, z) coordinates, (x, y, z) rotation, as well as the parent's rotation. The World Server keeps a hierarchical structure of objects. In order to calculate the absolute coordinates of an object, the world server starts by translating and rotating first the leaves of the object tree and moves up the tree to the root constantly applying the translate/rotate process. (This is not the actual algorithm however).

A single object is first translated to its relative (x, y, z) coordinates which the function **MoveObject** modifies and then the object is rotated around the Z axis, followed by a rotation around the X axis, followed by a rotation around the Y axis by the (x, y, z) rotation values which the function **RotateObject** modifies. The absolute coordinates of all the children of this object are subsequently modified.

Return Value

None.

See Also

MoveObject, RotateObject

SetTimer

Summary

Non-Blocking

```
void SetTimer(pConnection, hObject, milliseconds, seconds);
PCONNECTION    pConnection;
HOBJECT        hObject;
long           milliseconds, seconds;
```

Description

This function does not send any messages to the server but simply sets up a periodic timer interrupt that will periodically call the message handling function for the object specified by **hObject** with a TIMER message. The periodic function call is a direct call. The message is not placed in the message queue like all other messages arriving at the message handling function.

The values in **milliseconds** and **seconds** determine the rate of the periodic callback. The interval between to consecutive callbacks is given by adding the two values together.

Only one timer can be set per application.

Return Value

None.

See Also

CreateObject, StopTimer

StopTimer

Summary

Non-Blocking

void StopTimer(void);

Description

This function disables the automatic periodic callback mechanism established with the **SetTimer** function call.

Return Value

None.

See Also

SetTimer

DefineShape

Summary

Blocking

```
int DefineShape(pConnection, hObject, sName, sFormat);
PCONNECTION   pConnection;
HOBJECT       hObject;
char          *sName;
char          *sFormat;
```

Description

This function creates a format for the object identified by **hObject**. **sName** is a pointer to a string containing the name of the format you want to register. **sFormat** is a pointer to memory containing the text of the format itself.

The format body must be some zero-terminated ASCII text. Future versions of the system may allow the application to specify a format that is not necessarily ASCII.

The format of an object describes the 3D shape of that object. Examples of such descriptions are the DXF format, VOF format, NFF format, etc. The format of an object is uploaded to the world server by the application. Any viewer wishing to display the object must download this format by calling the **GetShape** function.

Upon return, memory that may have been allocated and pointed to by **sName** or **sFormat** can be deallocated.

Return Value

Returns a non-zero value if successful. NULL if error.

See Also

DefineShapeFromFile, GetShape, GetShapeFormat

GetShape

Summary

Blocking

```
char *GetShape(pConnection, hObject, sName, sFormat);
PCONNECTION     pConnection;
HOBJECT         hObject;
char            *sFormat;
```

Description

This function returns a pointer to memory containing the format whose name is given by the string at the address contained in **sName** of the object in **hObject**.

The format body returned by this function is a zero-terminated ASCII string. Future versions of the system may allow the application to specify a format that is not necessarily ASCII.

The format of an object describes the 3D shape of that object. Examples of such descriptions are the DXF format, VOF format, NFF format, etc. The format of an object is uploaded to the world server by the application by calling the function **DefineFormat**.

The memory returned by this function may be deallocated if no longer needed.

Return Value

Returns a pointer to a memory buffer containing the format of the object as a zero-terminated ASCII string. Returns NULL if error.

See Also

DefineShape, DefineShapeFromFile, GetShapeFormat

GetShapeFormat*

Summary

Blocking

```
void *GetShapeFormat(pConnection, hObject, sPrevFormat, sNextFormat);
PCONNECTION    pConnection;
HOBJECT        hObject;
char           *sPrevFormat;
char           *sNextFormat;
```

Description

This function retrieves in **sNextFormat**, the name of the next available format for the object specified by **hObject**. The format must have been registered with **DefineFormat** or **DefineFormatFromFile**.

sPrevFormat is a pointer to a string containing the name of the previously obtained format. If this is NULL or empty, the first available format is returned in **sNextFormat**.

The format of an object describes the 3D shape of that object. Examples of such descriptions are the DXF format, VOF format, NFF format, etc. The format of an object is uploaded to the world server by the application by calling the function **DefineFormat**.

Return Value

None.

See Also

DefineShape, DefineShapeFromFile, GetShape

DefineShapeFromFile*

Summary

Blocking

```
int DefineShapeFromFile(pConnection, hObject, sName, sFName);
PCONNECTION     pConnection;
HOBJECT         hObject;
char            *sName;
char            *sFName;
```

Description

This function creates a format for the object identified by **hObject**. **sName** is a pointer to a string containing the name of the format you want to register. **sFName** is a pointer to a string containing the name of the file containing the format data.

The format data must be some ASCII text. Future versions of the system may allow the application to specify a format that is not necessarily ASCII.

The format of an object describes the 3D shape of that object. Examples of such descriptions are the DXF format, VOF format, NFF format, etc. The format of an object is uploaded to the world server by the application. Any viewer wishing to display the object must download this format by calling the **GetShape** function.

Upon return, memory that may have been allocated and pointed to by **sName** or **sFName** can be deallocated.

Return Value

Returns a non-zero value if successful. NULL if error.

See Also

DefineShape, GetShape, GetShapeFormat